Garbage Collection for Memory-Constrained Java Virtual Machines*

Tarek S. Abdelrahman and Sirish R. Pande
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4

Abstract

We describe and evaluate a modified mark-and-sweep garbage collector for Java Virtual Machines (JVMs) that allows applications to run using less memory. The modified algorithm not only removes from the heap objects that are no longer needed by a program (i.e., garbage), but also objects that have not been accessed for a sufficiently long time. The removal of such *idling* objects affects the reachability of other objects on the heap, and introduces overhead to both execution time and space requirements of the JVM. We describe how these issues are addressed, and evaluate the modified collector in the Sun Microsystems JDK 1.2.2 using applications from the SPECjvm98 benchmark suite. Our results indicate that it is possible to execute applications with up to 20% less memory, but at the expense of longer execution time. Our modified algorithm is useful when an application is to execute in a memory-constrained environment, where it normally runs out of memory.

Keywords: Compiler and run time support; garbage collection; memory usage optimization.

1. Introduction

The use of Java in parallel programming has steadily increased since the introduction of the language in the mid 1990s [1, 3]. This can attributed to the several features of the Java programming language and its execution model. These features include: the integration of threading into the language specification, increased portability, and automatic memory management through garbage collection (GC). This paper is concerned with the design and evaluation of a new GC algorithm that aims to reduce the memory requirements of Java programs, compared to existing GC algorithms.

The majority of garbage collectors fall into the category of *mark-and-sweep* collectors. Such collectors operate in 3 phases to remove heap objects no longer needed by an executing program (i.e., garbage). In the first phase, the *root set* of the executing program is identified. This is the set of all variables (both local and global) that can be used to access heap objects by the executing program. In the second phase, all heap objects that are reachable (both directly and indirectly) via pointer references from the root set are identified and marked. In the last and final phase, a sweep is made of the heap to determine objects that are not reachable from the root set. Such objects are designated as garbage. Since the executing program has no means of accessing them, they are removed from the heap.

Recent studies [4, 5, 6] show that a large fraction of objects on the heap are not accessed again beyond a certain point in program execution, but remain reachable from the root set. These objects are not used by the executing program, yet cannot be claimed as garbage because they can potentially be used through one of the variables in the root set. Such objects are referred to as *drag* objects. They unnecessarily occupy memory, and hence, increase the memory usage of the program. This can become problematic if the program is to be executed on a memory-constrained Java Virtual Machine (JVM), where it normally runs out of memory.

In this paper, we present a new mark-and-sweep GC algorithm that moves dragged objects to secondary storage, thus allowing a JVM to run with tighter memory constraints. The basic idea behind the algorithm is to monitor the time an object has not been accessed. When this time exceeds a given threshold, the object is assumed to be drag, and is moved out of the heap and onto the disk. However, since there is no guarantee that the object will indeed not be accessed again, enough information is maintained in memory to retrieve the object if and when it is accessed. We implemented and evaluated the algorithm in the Sun Microsystems JDK 1.2.2. Our experimental results indicate that our GC algorithm increases execution time, but does enable benchmarks to run with up to 20% less memory.

The remainder of this paper is organized as follows. Section 2. provides some background material on GC and object drag. Section 3. describes our modified GC algorithm. Section 4. presents our experimental evaluation of the algorithm. Section 5. describes related work. Finally, Section 6. given some concluding remarks and directions for future work.

2. Background

In this section, we present background material necessary to understand our garbage collector. More specifically, we describe the structure of the heap and the overall operation of the mark-and-sweep garbage collector employed in our experimental testbed, the Sun Microsystems JDK 1.2.2. For simplicity, we will just refer to this testbed as the JVM. However, it should be noted that different JVMs could employ different heap structures and/or different garbage collectors. Nonetheless, our GC algorithm applies to those JVMs as well.

*This work was supported by research grants from NSERC and CITO.

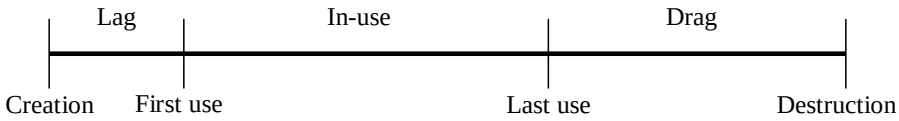


Figure 1. Phases in the lifetime of an object (after [4]).

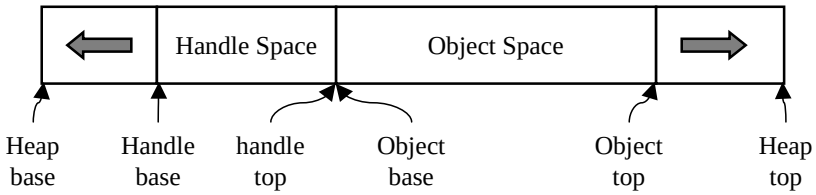


Figure 2. The layout of the heap in the JVM.

2.1 Object drag

The various phases in the lifetime of an object on the Java heap is depicted in Figure 1 (after [4]). The time interval between the the creation of an object on the heap and its first use is referred to as the lag time. The object is in use between the first use and the last use. The interval between the last use and the destruction (or garbage collection) of the object is referred to as the "drag" time. During its drag time, the object unnecessarily occupies memory. In general, it is not possible during program execution to determine if an object is drag, since it is not possible to determine if a use of the object is indeed the last use.

2.2 Heap structure

The heap space in the Sun Microsystems JDK 1.2.2 is divided into two major parts: a handle space and an object space. Every object on the heap is represented by a handle, which contains a pointer to the object itself in the object space and a pointer to the method table of the object. An object can be accessed only through its handle, by traversing the appropriate pointer in the handle. The use of a handle to access objects facilitates heap compaction and the relocation of objects in the object space. On the other hand, it introduces an additional pointer traversal for every object access.

The overall layout of the heap is shown in Figure 2. The JVM starts with initial handle and object spaces. It allows each to independently grow in opposite directions until a hard limit is reached (heap base and heap top, respectively). Thus, a Java program may run out of space for two reasons. First, it may allocate objects whose size exceeds the largest possible object space. Second, the program may use a large number of objects at the same time that it runs out of handles.

Objects are allocated on the object space using a next-fit allocation policy, which may lead to heap fragmentation. Thus, the JVM compacts the heap at garbage collection time to reduce fragmentation.

2.3 The mark-and-sweep garbage collector

The mark-and-sweep GC algorithm is depicted in Figure 3. The root set of the executing program is first generated by scanning global references, Java stacks and the native stack. The root set is then marked as reached. The objects on the heap are then scanned (using their handles). If a heap object is marked as reached, then its children through pointer references are also marked as reached. This process effectively generates a graph of objects that are reachable from the root set of objects. This graph is referred to as the *reachability* graph. Once all reachable objects have been marked (or equivalently, the reachability graph has been generated), all the objects on the heap are scanned once more. Objects that are not marked as reachable (i.e., not part of the reachability graph) are marked as garbage. Finally, garbage objects are removed from the heap and the object space part of the heap is compacted.

The GC is invoked when an attempt to allocate memory is made and no space is available. The threads of the executing program are stopped and a GC thread is started. This is referred to as *stop-the-world* garbage collection [2]. The GC thread executes the algorithm depicted in Figure 3, and then execution of the program threads is resumed.

3. The modified garbage collector

3.1 Overview

The basic idea behind our modified garbage collection algorithm is to monitor the idle time of objects on the Java heap and to remove an object from the heap once its idle time has exceeded some threshold, under the assumption that the object has become drag. Since idle objects are not necessarily garbage, and may indeed be accessed again, an idle object is removed to secondary storage (the disk) and information is left on the heap (as part of the handle to the object) to allow the retrieval of the object if and/or when necessary. This is depicted graphically in Figure 4.

A timestamp variable is added to the handle of each

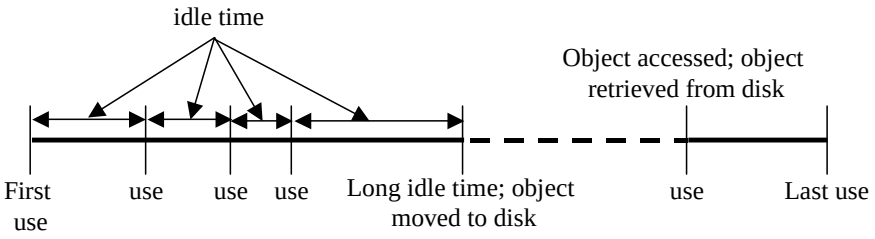


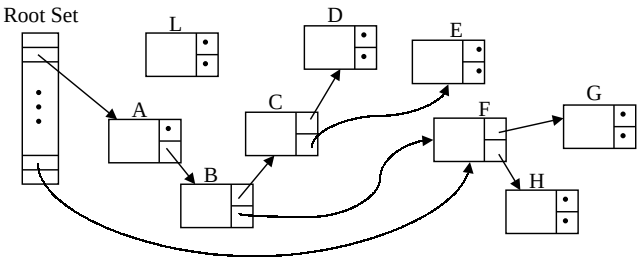
Figure 4. Phases in the use of an object.

```
Initialize()
Generate_root_set()

for each object R in the root set
   for each reference r in R
      mark target of r as reached
   end for
end for

// Mark phase
for each object O on the heap
   if (O is reached)
      for each reference r in O
        mark target of r as reached
      end for
   end if
end for

// Sweep phase
for each object O on the heap
   if (O is not reached)
      designate O as garbage
   end if
end for

// Cleanup
remove objects marked as garbage
Compact_heap()
```

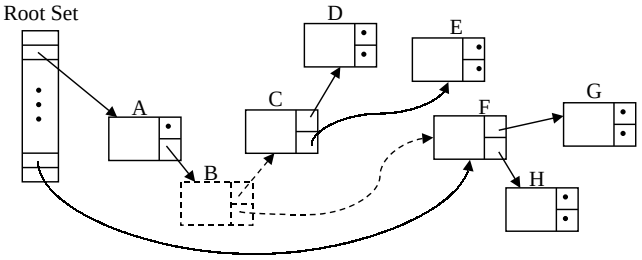
Figure 3. The mark-and-sweep garbage collector.

object. When an object is first created, this variable is initialized to the creation time. On each subsequent access to the object, the timestamp variable is assigned the time of access. When the garbage collector is invoked, the difference between the garbage collection invocation time and the timestamp variable value defines the amount of idle time of each object. During the mark phase of the garbage collector, the idle time of each object on the heap is calculated. During the sweep phase of the collector, objects whose idle time has exceeded the threshold are moved to the disk.

The removal of idle but reachable, and hence potentially accessible, objects from the heap has an impact on object reachability, on execution time, and on space.



(a) An example reachability graph after the mark phase.



(b) The example reachability graph after the sweep phase.

Figure 5. The effect of idle object removal on the reachability graph.

3.2 Impact on object reachability

In general, a removed idle object is potentially accessible, and is thus part of the reachability graph, which is implicitly built by the garbage collector during its mark phase. Removing such an object may disconnect the reachability graph, resulting in components that are not reachable from the root set, yet whose nodes correspond to reachable objects. In a subsequent garbage collection, such nodes would be considered garbage, and would incorrectly get deleted from the heap.

This situation is illustrated in Figure 5(a), which shows an example reachability graph of objects on the heap, as determined at the end of the mark phase of a garbage collection. Object L is clearly not reachable from the root set, and thus would be removed from the heap duing the sweep phase of the same garbage collection. Assume that object B has idled sufficiently long, and is removed from the heap during the sweep phase. The subgraph formed by objects C, D, and E is now disconnected from the rest of the graph, as shown in Figure 5(b), which depcits the reachability graph

at the end of the sweep phase of the same garbage collection. These three objects are now not reachable from the root set. Consequently, a subsequent garbage collection will determine that C, D and E are not reachable, and will remove them as garbage. Yet, those objects are potentially accessible, reachable through B, which is now on disk. Thus, a mechanism is needed to distinguish such disconnected but potentially accessible objects from garbage to avoid their eventual collection.

Note that although the removal of B also removes the path from the root set of F, G, and H through B, it does not render these objects disconnected from the reachability graph. This is because there exists another path from the root set to these objects through the root set.

In order to address this problem, we flag objects that are descendants of an idle object that is to be moved to disk. We refer to these children as *sticky* objects. Such objects are not removed from the heap during a garbage collection, even when they are not reachable from the root set. This is because they descend (directly or indirectly) from an object on disk, which itself is reachable.

The modified GC algorithm is depicted in Figure 6. The root set of the executing program is generated as in the original GC algorithm. The idle time of each object on the heap is calculated, and objects that are to be moved to disk are flagged as idle. During the mark phase, objects directly reachable from idle objects are marked as sticky, since idle objects will be removed from memory to disk. Children of sticky objects are also marked as sticky. Only objects that are on the heap are used to determine the reachability of other objects. In other words, objects moved to disk are not considered when determining the reachability of other objects on the heap. In the sweep phase, idle objects are moved to disk. Also, sticky objects are not removed, even if they are not reachable.

3.3 Impact on execution time

Our modified GC algorithm incurs a number of overheads that impact execution time. These overheads stem from a number of sources in the modified JVM:

- *timestamping*. The modified GC algorithm maintains the time at which an object was last accessed. This requires that on every object access the JVM obtains and records the time of access. This represents overhead incurred by the modified JVM compared to the original one.

- *in-memory check*. The modified JVM requires that on every object access the presense of the object in memory be checked. The time taken to check the in-memory flag stored on the handle represents overhead incurred by the modified JVM compared to the original one.

- *garbage collection*. The modified GC algorithm performs additional steps compared to the original mark-and-sweep collector. For example, the modified GC algorithm must calculate the age of each object on the heap, move sufficiently-long idling objects to disk, and mark the descendants of moved objects as sticky. These steps represent overhead that is not incurred by the original JVM, but is incurred by the modified one.

```
Initialize()
Generate_root_set()

for each object R is root set
   for each reference r in R
      mark target of r as reached
   end for
end for

for each object O on the heap
   idle time = GC_time-timestamp
   if (idle time > threshold)
       flag object as idle
   end if
end for

// Mark phase
for each object O on the heap
   if (O is reached)
      for each reference r in O
        if (target of r is in memory)
           mark target of r as reached
        end if
      end for
   end if

   if (O is idle or O is sticky)
      for each reference r in O
        if (target of r is in memory)
           mark target of r as sticky
        end if
      end for
   end if
end for

// Sweep phase
for each object O on the heap
   if (O is idle)
      move O to disk
   end if

   if ((O is not reached) and
       (O is not sticky))
         designate O as garbage
   end if
end for

// Cleanup
remove objects marked as garbage
Compact_heap()
```

Figure 6. The modified mark-and-sweep garbage collector.

Further, because the modified JVM is executing with a smaller heap size, the GC is invoked more frequently in the modified JVM compared to the original one. The time for additional garbage collector invocations represents overhead.

- *object retrieval*. The modified JVM must retrieve an object from disk if the object is accessed. The time to retrieve objects is overhead that is incurred by the modified JVM compared to the original one.

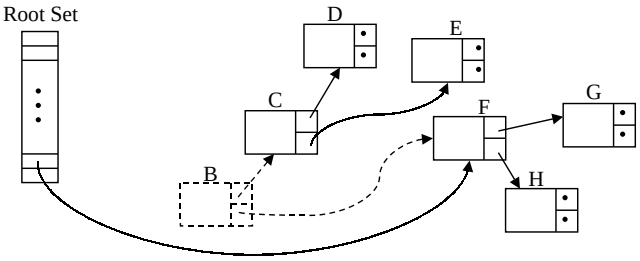


Figure 7. An example reachability graph after the sweep phase.

Benchmark	Time (sec)	Size (Mbytes)
compress	771	36.9
jess	181	2.1
raytrace	312	6.4
db	383	11.8
mpeg	642	4.7
jack	245	2.6

Table 1. The benchmark applications, their execution time, and their minimum heap sizes.

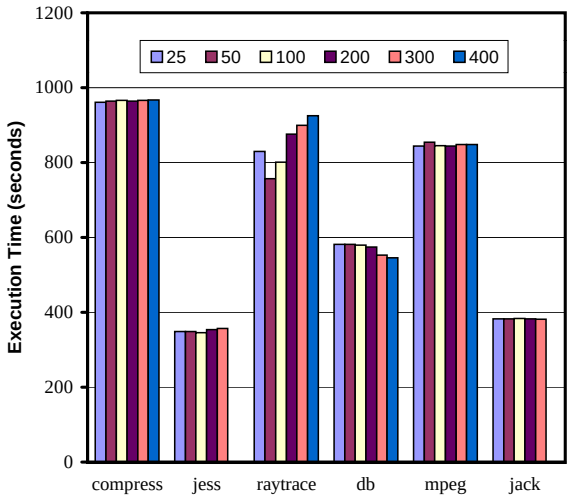


Figure 8. The impact of the user-specified idleness threshold on the execution time of the application benchamrks.

3.4 Impact on space

The removal of potentially accessible objects from the heap and the introduction of sticky objects gives rise to space overhead. This overhead basically stems from the fact that once an object becomes sticky, it is not garbage collected, even when it does become garbage. This can be illustrated using the example reachability graph shown earlier in Figure 5(b). Object B, which has been moved to the disk is shown in dotted lines, and is only reachable through its parent, object A. Objects C, D, and E are sticky. If object A is deleted, this makes object B garbage, as shown in Figure 7. However, since B is on disk, it is not considered during the mark phase of the modified GC algorithm, and objects C, D, and E are not recognized as garbage.

While sticky garbage objects consume space, such objects eventually idle long enough and are removed from the heap, recovering their space. Nonetheless, when they are moved to the disk, their handles remain in the handle space and contribute to space overhead.

4. Experimental evaluation

We modified the Sun Microsystems JDK 1.2.2 to implement our modified GC algorithm. We used the modified JVM to execute applications from the SPECjvm98 benchmark suite [7]. We compare the performance of these application on the modified JVM to their performance using the original JVM. Both versions of the JVM were executed on a dedicated 500-Mhz UltraSPARC-IIe workstation with 384 MBytes of memory and running Solaris v2.8. Each benchmark application is executed 4 times, and the average execution time is reported.

In order to simulate a constrained memory environment we executed the benchmarks using both the original JVM and the modified JVM so that each benchmark uses the smallest possible heap size. That is, it is not possible to execute an application in a smaller heap, and indeed the application terminates with an "out-of-memory" error. Thus, if a benchmark executes with less memory using our modified GC algorithm, then our algorithm is successful in allowing the benchmark to run in a more memory-constrained environment. Table 1 lists the benchmark applications, their execution time, and the corresponding minimum heap size used to execute each benchmark on the original JVM.

In the remainder of this section, we present our evaluation of the modified JVM using four sets of results. The first set describes the impact of the idleness threshold on execution time. The second set shows the impact of heap misses on execution time. The third set gives the overall saving in space achieved by the modified GC algorithm, and also describes the impact of the space overheads described in Section 3.4. The fourth and final set of results shows the overall execution time of the modified JVM, including breakdown and analysis of the various sources of performance overhead discussed earlier in Section 3.3.

Figure 8 depicts the overall execution time of each application on the modified JVM for different values of the user-specified idleness threshold (in seconds). The figure shows that the performance of `compress`, `jess`, `mpeg`, and `jack` is largely insensitive to the threshold value. In contrast, the performance of `raytrace`, and to a lesser extent `db` varies with the value of the user-specified threshold. However, the amount of variation is not significant Thus, we elect to present our results in the remainder of this section with a user-specified idleness threshold value of 100 seconds.

Table 2 gives the heap miss rate m_{heap} for each of the applications, which is defined as the number of accesses to objects that have been removed from the heap to the total number of object accesses. The table also gives the miss penalty associated with these misses, which is the time taken

Benchmark	Miss Rate	Miss Penalty
compress	1.1e-8	0.0%
jess	1.3e-5	0.4%
raytrace	2.0e-5	0.6%
db	1.1e-4	1.5%
mpeg	1.9e-8	0.0%
jack	2.9e-5	0.6%

Table 2. The heap miss rate and miss penalty for the benchmarks.

to retrieve objects from disk expressed as a percentage of the overall execution time of the original JVM. The table indicates that m_{heap} for each application is very small. Furthermore, the penalty of misses is relatively small–less than 1.5% of the original execution time. This validates our object removal heuristic; the probability of accesses to objects that have remained unaccessed for a sufficiently long time on the heap is low. Further, the penalty associated with heap misses has little impact on performance.

Table 3 gives the minimum memory space needed by each of the benchmarks on both the original and the modified JVMs. The table gives the size of handle space, the size of the object space, and the total size of the heap space for each benchmark. In the case of the original JVM, the total size of the heap is the sum of the handle and object space sizes. However, in the case of the modified JVM, the total size of the heap is larger than the sum of the handle and object space sizes. This is because of the need for additional memory to store the timestamp and various flags. While the timestamp and flags are conceptually part of the handle, they are stored outside the handle for efficiency of implementation. The table also shows the percentage gain (i.e., reduction) in space for each of handle, object and heap spaces.

The table indicates that the modified GC algorithm achieves space saving in the object space for all the applications except "jack". The savings range from 2.3% to 38.0%. However, the overall saving in heap space is offset by an increase in the handle space, which is due to sticky objects. Such handle space increase can be significant (as in cases of "jack" and "jess") such that the net effect is an overall increase in heap space requirements.

Figure 9 depicts the overall execution time of the modified JVM, normalized with respect to the execution time of the original JVM. Thus a greater-than-one normalized execution time indicates a slow-down in execution. The figure shows that all applications suffer from a slow down, ranging from 1.25 to 2.5.

This slow down is attributed to the various sources of overhead that were described in Section 3.3. In order to measure the impact of a particular source of overhead, the original JVM is modified to include only actions that lead to this source of overhead. The execution time of this modified JVM is then compared to that of the original JVM to compute the impact of the overhead on execution time. This method of measuring overhead is preferred to fully and simultaneously instrumenting the modified JVM to obtain the impact of all sources overhead at once, due to the intrusive nature of the instrumentation calls, which is likely to skew

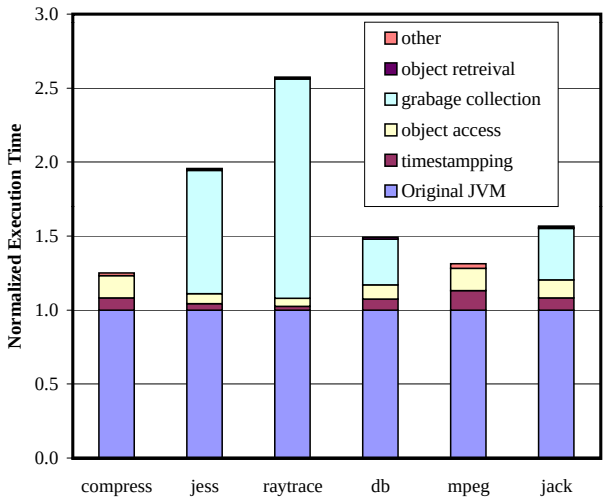


Figure 9. The execution time of the application benchamrks on the modified JVM, normalized with respect to the original JVM.

the results.

Figure 9 also shows the contribution of each type of overhead to the execution time of the modified JVM. The figure shows that the most significant source of overhead is that of garbage collection. Not only is the time for each garbage collection higher, but also the number of GC invocations is significantly higher, as indicated in Table 4. This is mainly due to the space overhead in the handle space, which was described in Section 3.4. As the remaining handle space becomes smaller, the GC is invoked more often to free more handles.

5. Related work

Shaham et al. [5] conduct a profiling study of objects on the Java heap. They conclude that, for the SPECjvm98 suite, heap objects that remain reachable on the heap but are not used consume 23% to 74% of the heap space. In follow up study [6] they design a heap profiling tool to explore the potential for memory savings in Java programs. Our work builds on the results of their work, and extends it by presenting a modified GC algorithm that removes from the heap objects that are reachable but no longer used by the program.

Our modified GC algorithm bears resemblance to generational garbage collectors that move "aging" objects to a separate area in memory, which is not scanned every garbage collection [2]. The goal of generational collectors is to reduce collection time by exploiting the empirical observation that young objects are much more likely to be garbage collected than old objects. Generational collectors do not remove aging objects from memory, but use *remembered sets* to capture references from old objects to younger ones. In contrast, our work aims to reduce the amount of memory used by an application, and removes from memory objects that have been sufficiently idle, which is largely independent of the age of an object (i.e., old objects may still be accessed frequently, while younger objects may go

Benchmark	Original JVM			Modified JVM			% Gain in Space		
	Heap	Handle	Object	Heap	Handle	Object	Heap	Handle	Object
compress	36.9	7.4	29.5	29.3	7.3	18.3	20.6%	0.8%	38.0%
jess	2.1	0.4	1.6	2.4	0.8	1.3	-18.7%	-90.6%	22.3%
raytrace	6.4	1.3	5.1	5.6	1.5	3.3	12.1%	-21.2%	35.5%
db	11.8	2.4	9.4	10.6	2.4	7.0	10.0%	0.0%	26.1%
mpeg	4.7	0.9	3.8	4.5	0.5	3.7	5.4%	45.5%	2.3%
jack	2.6	0.5	2.1	5.5	1.9	2.6	-114.2%	-283.6%	-25.1%

Table 3. The minimum memory space (in Mbytes) needed to run the benchmarks on the original and modified JVMs. The table shows the minimum handle and object spaces as well as the total heap size.

Benchmark	Original		Modified	
	# Invocations	Time/GC	# Invocations	Time/GC
compress	17	13.1	37	33.2
jess	771	33.3	2773	63.7
raytrace	1518	67.5	2735	206.4
db	469	109.5	1001	169.3
mpeg	4	4.5	12	7.5
jack	505	16.9	1554	60.2

Table 4. The number of GC invocations and average time (in milliseconds) per GC on the original and modified JVMs.

idle, or vice versa). Further, remembered sets occupy additional memory space. Although sticky objects also occupy additional memory space when they become garbage, they idle and are eventually are removed to disk. Thus, unlike remembered sets, they occupy additional memory space only for a limited period of time.

6. Concluding remarks

In this paper we presented a modified GC algorithm that allows applications to run with smaller memory space on JVMs. The algorithm removes from the heap objects that have not been accessed for a sufficiently long period of time, freeing their memory space. However, since such objects are not garbage, they are potentially accessible. Thus, information is left in memory to retrieve these objects from secondary storage if/when necessary. The algorithm tackled a number of issues that result from the removal of such potentially accessible objects.

Our experimental evaluation validates our heuristic that long idling objects are rarely accessed again; the heap miss rate is very small, and the resulting penalty is less that 1.5% of overall execution time. Our evaluation also indicates that space savings can be realized, allowing applications to run in up to 20% smaller memory, albeit at the expense of higher execution time. The space saving in object space is higher, but the overhead in handle space reduces the space saving. We believe that the algorithm is useful in memory-constrained environments, where an application may not run with the amount of available memory, but will run with our modified GC algorithm.

Future work will focus on mechanisms for limiting or eliminating handle space overhead introduced by sticky garbage objects, possibly at the expense of higher execution time overhead. Future work will also examine means to reduce the space consumed by garbage objects on disk.

References

[1] B. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison Wesley, 1996.

[2] R. Johns and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, 1996.

[3] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison Wesley, 1997.

[4] N. Rojemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. of Int'l Conf. on Functional Programming*, pages 34–41, 1996.

[5] R. Shaham, E. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *Proc. of Int'l Symp. on Memory Management*, pages 12–17, 2000.

[6] R. Shaham, E. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proc. of Prog. Language Design and Implementation*, pages 104–113, 2001.

[7] SPECjvm98, http://www.specbench.org/ osg/jvm98, 1998.